Implementing Relational Operators: Selection, Projection, Join

Readings

- [RG] Sec. 14.1-14.4

Last time

- Started discussion on how to implement RA operators (selection)
- Metric: #I/Os required
  - don't include the cost of writing out results
  - same for all implementations

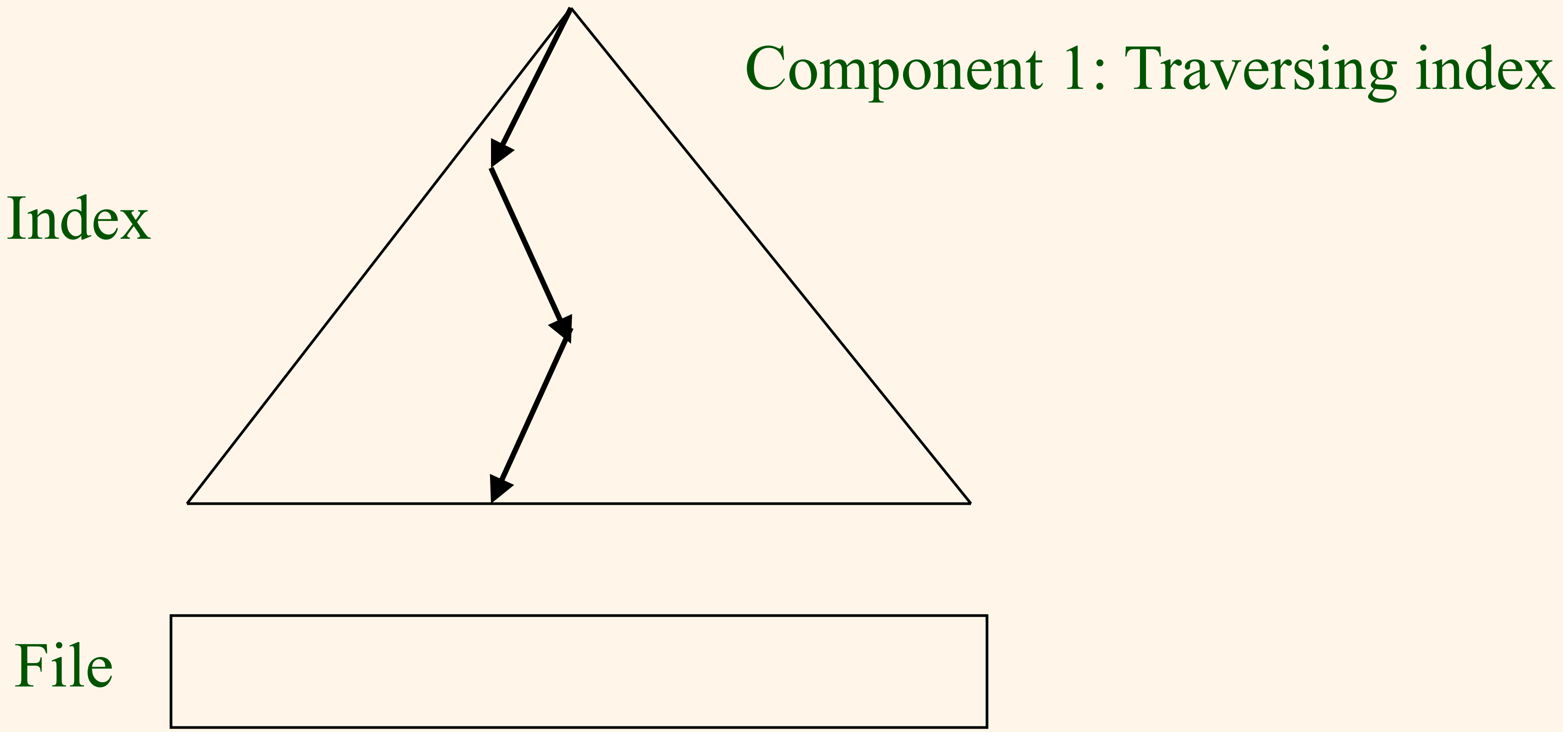
No index

- Unsorted file (or sorted on "wrong" attribute)
  - Only option is to scan whole thing linearly
  - Cost: # pages in the file
- File is sorted on selection attribute
  - Binary search, then retrieve all qualifying tuples
  - Cost: log (#pages in file) + #pages with qualifying tuples

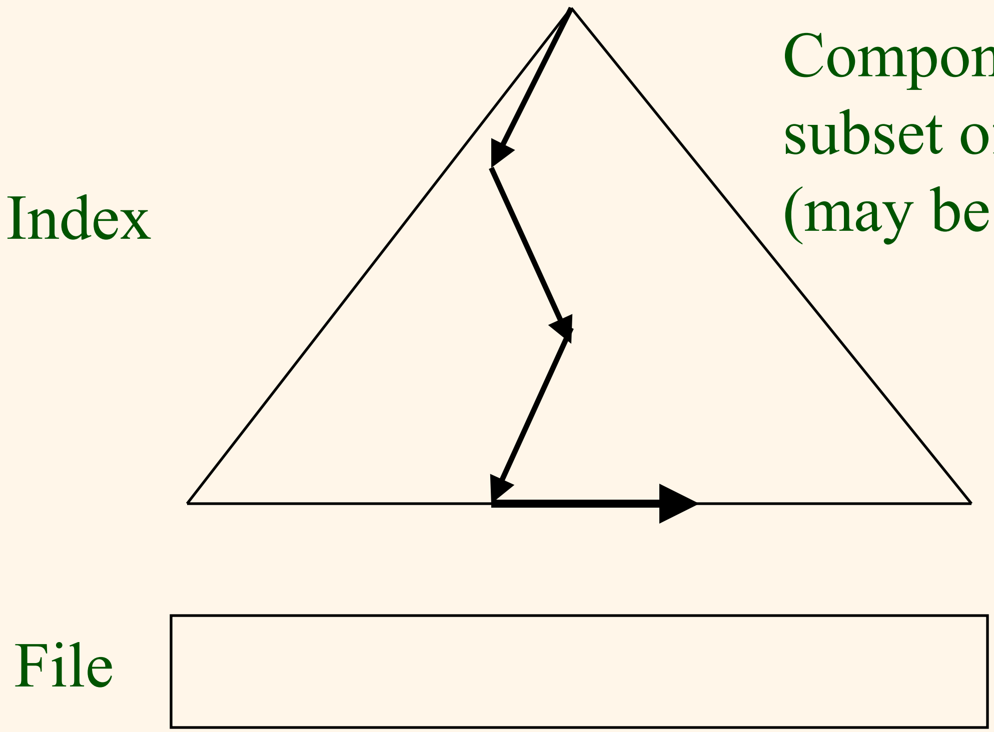
Using indexes for selection

- Tree index cost:
  - Traverse tree from root
    - Done only once
    - 2 to 3 I/Os (trees are short)
  - Scan leaf level pages to find all data entries
  - Retrieve actual data records

Cost Components – Tree Index



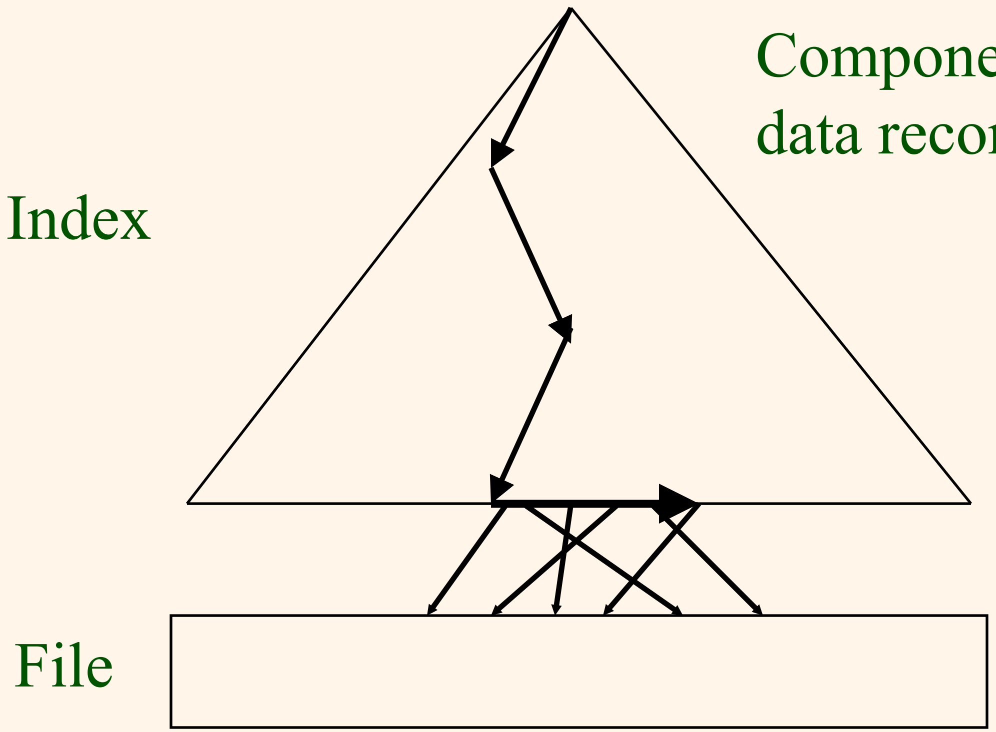
Cost Components – Tree Index



Component 2: Traversing subset of data entries in index (may be ok to skip if clustered)

Cost Components – Tree Index

Component 3: Fetching actual data records (alternative 2 or 3)



What about hash indexes?

- Depends on implementation (linear, extendible etc)
- But typically small
- Reasonable assumption: 1 or 2 I/Os to find right bucket
  - Plus cost of retrieving actual data records (depends on number of matching records)

Example

- Selection on Reserves, condition is "R.rname < 'C%'"
- Assume uniform distribution of names, so about 10% of relation should be retrieved
  - 10K tuples, 100 pages

Example

- Have clustered B+ tree index on rname
  - Traverse index to find first leaf page - 1 or 2 I/Os
  - Start scan and retrieve tuples - 100 I/Os
- Have unclustered B+ tree index on rname
  - Worst case, retrieving each tuple requires a separate I/O so 10K I/Os
  - Much cheaper to just scan all 1000 original pages!
- Heuristic: scan cheaper than unclustered index if expect to retrieve more than 5% of tuples

Optimizations are possible

- Alternative 2 or 3, unclustered index
- Find qualifying data entries from index
- Sort the rids of the data entries to be retrieved
  - Remember rid = (page ID, slot #)
- Fetch rids in order
  - Ensures each data page is read from disk just once!
  - Although number of data pages retrieved still likely to be more than with clustering

Example

```
SELECT  *
FROM    Sailor S
WHERE   S.Age = 25 AND S.Salary > 100K
```

- Have Hash index on Age

Evaluation Options

- Option 1
  - Use available index (on Age) to get superset of relevant data entries
  - Retrieve the tuples corresponding to the set of data entries
  - Apply remaining predicates on retrieved tuples
  - Return those tuples that satisfy all predicates
- Option 2
  - Sequential scan! (always available)
  - May be better depending on selectivity

And another example...

```
SELECT  *
FROM    Sailor S
WHERE   S.Age = 25 AND S.Salary > 100K
```

- Have Hash index on Age
- Have B+ tree index on Salary

Evaluation Options

- Option 1
  - Choose most selective access path (index)
    - Could be index on Age or Salary, depending on selectivity of the corresponding predicates
  - Use this index to get superset of relevant data entries
  - Retrieve the tuples corresponding to the set of data entries
  - Apply remaining predicates on retrieved tuples
  - Return those tuples that satisfy all predicates

Evaluation Alternatives

- Option 2
  - Get rids of data records using each index
    - Use index on Age and index on Salary
  - Intersect the rids
    - We'll discuss intersection soon
  - Retrieve the tuples corresponding to the rids

- Option 3
  - Sequential scan!

More complex selection conditions

- When can we use an index?
- When it "matches" at least some of our selection condition.
- E.g. suppose we have a tree index for R on <sid, bid, day>
  - Will help for for "sid > 10"
  - Will help for "sid > 10 AND bid = 100"
  - But not helpful for "bid = 100"

Using indexes for selection

- A hash index *matches* (a conjunction of) terms that has a term *attribute = value* for every attribute in the search key of the index.

E.g., Hash index on <*a, b, c*> matches *a*=5 *AND b*=3 *AND c*=5; but it does not match *b*=3, *or a*=5 *AND b*=3, *or a*>5 *AND b*=3 *AND c*=5.

Using indexes for selection

- A tree index <u>*matches*</u> (a conjunction of) terms that involve only attributes in a *prefix* of the search key.

  E.g., Tree index on <*a, b, c*> matches the selection *a*=*5 AND b*=*3*, and *a*=*5 AND b*>*6*, but not *b*=*3*.

Complex Selections

(day<8/9/94 AND *rname='Paul')* OR *bid=5* OR *sid=3*

- Selection conditions are first converted to *conjunctive normal form* (CNF):

  (day<8/9/94 OR *bid=5* OR *sid=3)* AND
  (rname='Paul' OR *bid=5* OR *sid=3)*

Complex Selections

- Combination of techniques seen so far
  - If have index for one (or more) of the conjuncts, can use it to filter tuples and apply remaining condition to only those
  - Can use union of retrieved tuples (or RIDs) for "OR"
  - But really, at some point sequential scan becomes best bet

Selection summary

- Depends on what is available
  - File sorted on selection attribute(s)
  - Indexes (clustered or not)
- Options:
  - Sequential scan (not always totally stupid)
  - Binary search
  - Use index or combination of indexes

Projection

```
SELECT  S.Name, S. Age
FROM    Sailors S
```

```
SELECT  DISTINCT S.Name, S. Age
FROM    Sailors S
```

Implementing Projection

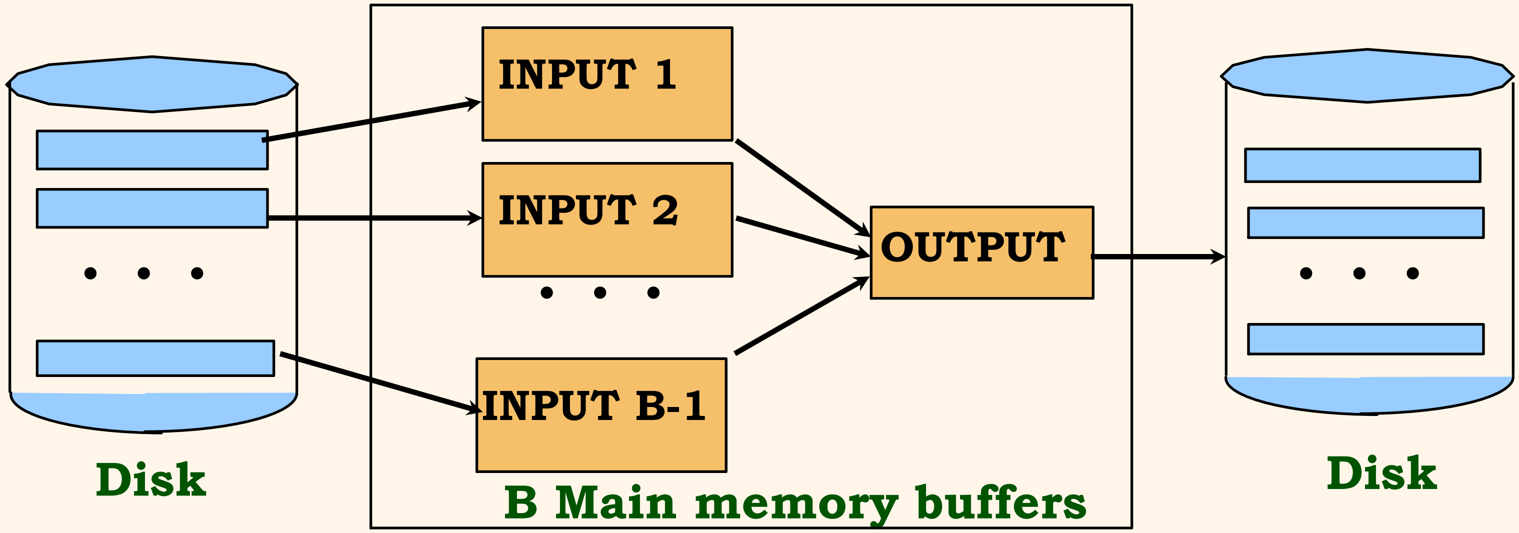
- In the general case, will need to scan the whole file
  - Because we want something from each tuple in the result
- The expensive part is eliminating duplicates if desired

Sorting-based projection

- Basic idea:
  - make a pass through tuples and retain only desired attributes
  - sort result of the above
  - go through resulting file and output only non-duplicates
- Fortunately, we know how to do external sorting for good performance!

General External Merge Sort

- Make multiple passes to merge runs
  - Pass 1: Produce runs of length B(B-1) pages
  - Pass 2: Produce runs of length $B(B-1)^2$ pages
  - …
  - Pass P: Produce runs of length $B(B-1)^P$ pages



Modifications to External Sorting

- Pass 0
  - Project out unwanted columns here (so don't need a separate pass before)
  - Still produce runs of length B pages
  - Tuples in runs are smaller than input tuples
- Merge passes
  - Eliminate duplicates during merge

Hashing-based projection

- Idea: hash all tuples into buckets based on attributes we want
- Bucket = partition (subset) of the input
- All duplicates will go into the same bucket

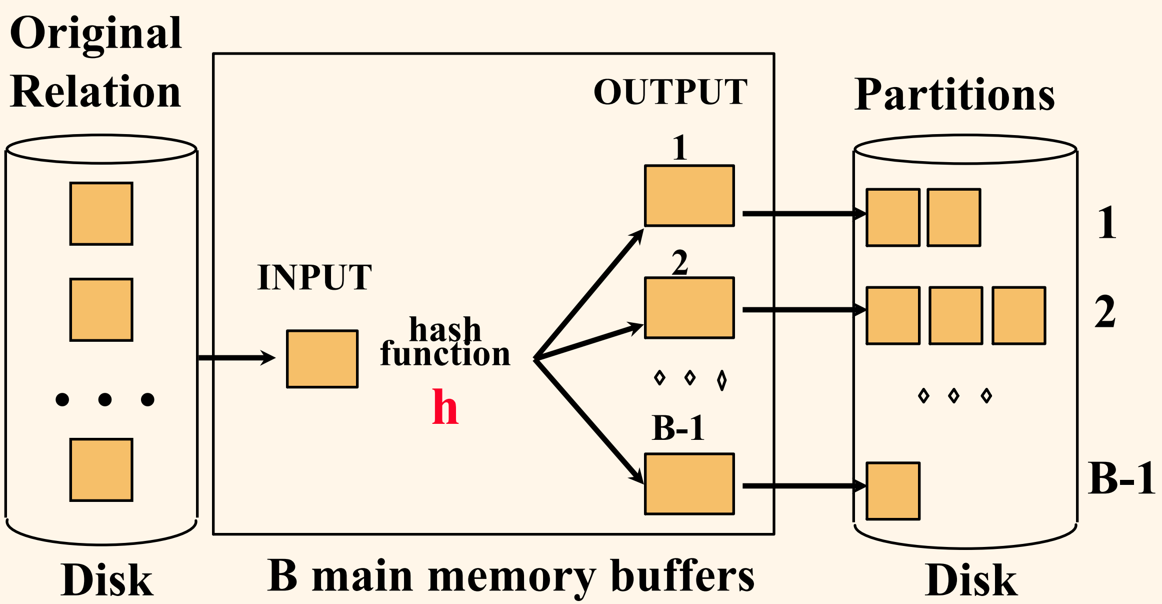
- Problem: hash table may not fit in memory!

Hashing-based projection

- If whole hashtable won't fit into memory, maybe a single bucket will
- All (pairs of) duplicates will be in the same bucket
- So we can eliminate duplicates bucket by bucket

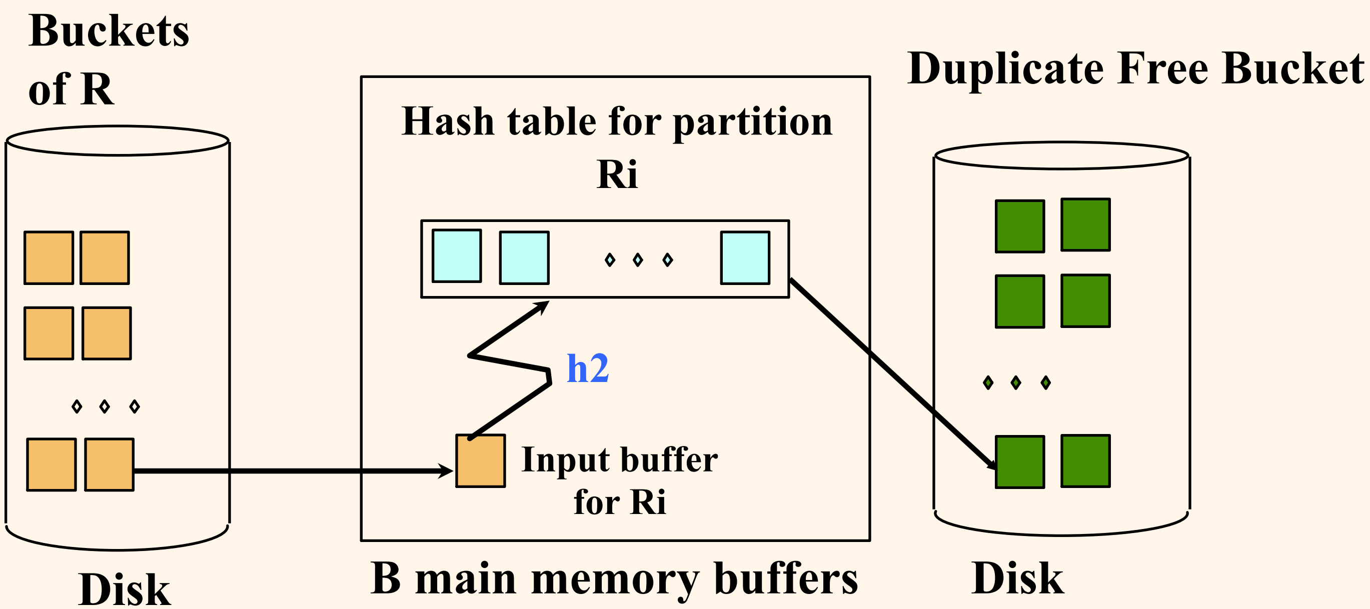
Projection Based on Hashing

- Assume relation does not fit in memory
- First pass
  - Divide relation into partitions
  - Eliminate unwanted fields as you go
  - No duplicate elimination yet!



Now process buckets one at a time

- Use a different hash function h2



- If hash table for bucket does not fit in memory, can recursively apply hashing algorithm from previous phase.

Comments on Projection

- Advantages of sort-based projection
  - Better handling of skew
  - Results in sorted order
- Hash-based projection lends itself better to parallelizing

Projection

```
SELECT DISTINCT S.Name, S. Age
FROM   Sailors S
```

- Have B+ tree index on (Name, Age)

Evaluation Using "Covering" Index

- Simply scan leaf levels of index structure
  - No need to retrieve actual data records
  - Index-only scan
- Works so long as the index search key includes all the projection attributes
  - Extra attributes in search key are okay
  - Best if projection attributes are prefix of search key
    - Can eliminate duplicates in single pass of index-only scan

Example

SELECT DISTINCT S.Name, S. Age
FROM Sailors S

- Have Hash index on Name
- Have B+ tree index on Age
- Sailors relation has 100 other attributes!

Evaluation Using RID Joins

- Retrieve (SearchKey1, RID) pairs from first index
- Retrieve (SearchKey2, RID) pairs from second index
- Join these based on RID to get (SearchKey1, SearchKey2, RID) triples
  - We will discuss joins soon!
- Project out the third column to get the desired result

Projection Summary

- General case: scan and eliminate duplicates
  - Sorting and hashing approaches
- Indexes: can use those if available
  - Index-only scans
  - RID joins

Next up: Joins!

SELECT *
FROM Reserves R, Sailors S,
WHERE R.sid = S.sid

- No indices on Sailors or Reserves

Tuple Nested Loop Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if r.sid == s.sid  then add <r, s> to result
```

- R is "outer" relation
- S is "inner" relation

Example relations

- Sailors (sid, sname, rating, age)
- Reserves(sid, bid, day, rname)

- Each tuple of Sailors 50 bytes long (80 tuples per page) and have 500 pages
- Each tuple of Reserves 40 bytes long (100 tuples per page) and have 1000 pages

Analysis

- Assume
  - M pages in R, p_R tuples per page
    - $M = 1000$, $p_R = 100$
  - N pages in S, p_S tuples per page
    - $N = 500$, $p_S = 80$
- Total cost = $M + p_R * M * N$
  - Ignore cost of writing out result
  - Same for all join methods
- This is 50,001,000 I/O's in our example!

Page Nested Loop Join

```
foreach page p1 in R do
    foreach page p2 in S do
        foreach r in p1 do
            foreach s in p2 do
                if r.sid == s.sid  then add <r, s> to result
```

- R is “outer” relation
- S is “inner” relation

Analysis

- Assume
  - M pages in R, p_R tuples per page
    - M = 1000, p_R = 100
  - N pages in S, p_S tuples per page Select
    - N = 500, p_S = 80
- Total cost = M + M * N
  - Which is 501,000 I/O s
- Note: Smaller relation should be "outer"
  - Better for S to be "outer" in this case!